A transaction approach to error handling

Bruce A Rafnel, Hewlett-Packard Co

You can apply the transaction-based recovery concept used in databases to any application. Doing so helps provide more reusable and maintainable programs.

Programs contain two major paths: a forward path that does the work and a reverse path that rolls back the work when the program detects errors. Typically, these paths are so tightly bound together that both paths are difficult to read. Code that is difficult to read results in code that is difficult to write, debug, enhance, and reuse.

For example, in object-oriented programming, you cannot reuse objects as much as you might want, primarily because the objects are tightly bound together at the error-handling level. Many times, error code even gives clues about how a program implements an object.

The solution is to handle errors in programs as you would in a database-transaction recovery mechanism. A database transaction is a unit of work that involves one or more operations on a database. For example, the operation of inserting data into a database could be a transaction if it's the only operation performed. If you combine the insertion with an update, the program considers both operations as one transaction. In a database transaction, the transaction either executes in its entirety, or, if an error appears in any of its operations, the transaction totally cancels as if it had never executed. If an error appears, the program automatically rolls back all work to the beginning of the transaction.

When a development team first introduces transaction error handling to a project, many engineers resist it because it requires the removal of IF statements after calls to functions. Engineers also believe the technique makes debugging more difficult. However, after seeing how much easier it is to read and write transaction error-handling code, the resistance fades. In addition, transaction error handling decreases debugging time to a little less than that of the traditional method. The reason for this decrease is probably that transaction error handling has less embedded error-handling code, causing defects to stand out more. Also, when you add error-handling code, you do so in the structured way that most engineers like to work—a method that disturbs very little of an already-debugged program.

BBS

The software listings in this article are available on *EDN*'s computer bulletin-board system. Phone (617) 558-4241 with modem settings 300/1200/2400 8,N,1. Access */freeware* SIG and specify *(r)ead* option followed by *(k)eyword* search for "MS #769."

Software developers are often dismayed at how difficult commercial programs are to maintain and design, compared with programs they developed in school. The reason for this may be that the programs students develop in school are "toys" that assume perfect inputs and that the hardware has unlimited memory and disk space. In addition, most software engineers have very little formal training in error-handling methods. Typically, software developers learn error handling by example or by trial and error, and they use the traditional error-handling model: Check for an error, find an error, and return an error code.

Many formal design processes, such as structured analysis and structured design, recommend that developers ignore errors during design because such errors are an implementation detail. However, this "minor" detail can take up to one-third of the code in commercial programs. This code appears not just around algorithms but directly in the middle of the algorithms. The resulting programs are difficult to read, debug, and reuse.

In addition to existing design methodologies, such as structured analysis and structured design, is exception handling, or error handling. This programming style separates most of the error-handling processes from the main algorithms. Error handling comprises four main parts: detection,

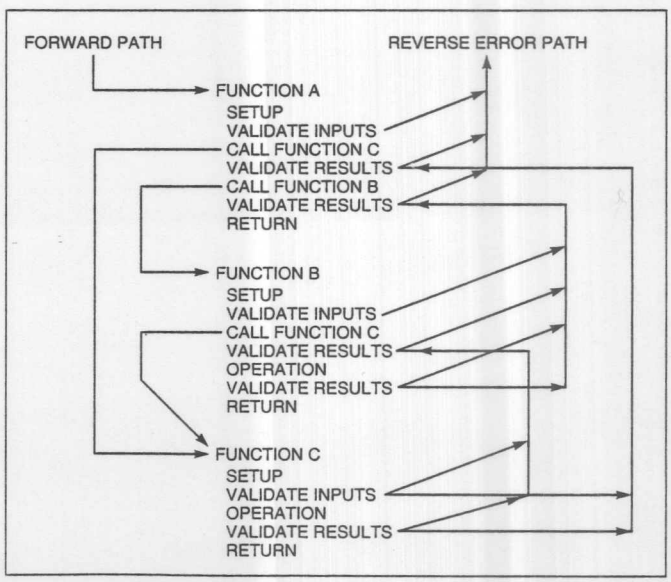

Fig 1—In traditional error-handling program flow, the forward path does the work of the program, and the reverse path does the error handling. Error-handling code appears throughout the algorithm.

correction, recovery, and reporting. The main focus of this article is error recovery.

In this context, the term "error" does not refer to a defect but to an exception that an algorithm cannot handle. A "defect," on the other hand, is an error that strains the design limits of an entire application or a system. For example, many algorithms assume that there is unlimited memory. Insufficient memory for the algorithm to complete successfully constitutes an error, and you must design the whole application to handle these out-of-memory errors. A defect occurs when an application that does not handle these errors causes a program to halt or to behave in an undocumented way. In other words, whether something is an error or a defect depends on what level of the software hierarchy you are observing.

Mixed forward and reverse path problem

Fig 1 shows the two major paths in commercial programs. The forward path does the work for which a program is designed. The reverse path is the error-handling code that keeps the forward path working correctly. It does this by detecting and fixing problems and rolling back partially completed work to a point at which the algorithm can again continue forward.

An intermediate function in a program has to stop what it is doing in the middle of the algorithm because the program called a function that cannot complete its task. This can lead to "tramp errors," a term similar to the "tramp-data" term of structured analysis and structured design (**Ref 1**).

Tramp errors in functions do not directly relate to the current function but are the result of a real error occurring in a lower-level function. For example, *function A()* calls *function B()*. *Function B()* needs some memory, so it calls the *malloc()* memory-allocation function. The *malloc()* function returns an out-of-memory error. This is a real error for the *malloc()* function. *Function B()* does not know how to get more memory, so it has to stop and pass the error back to *function A()*.

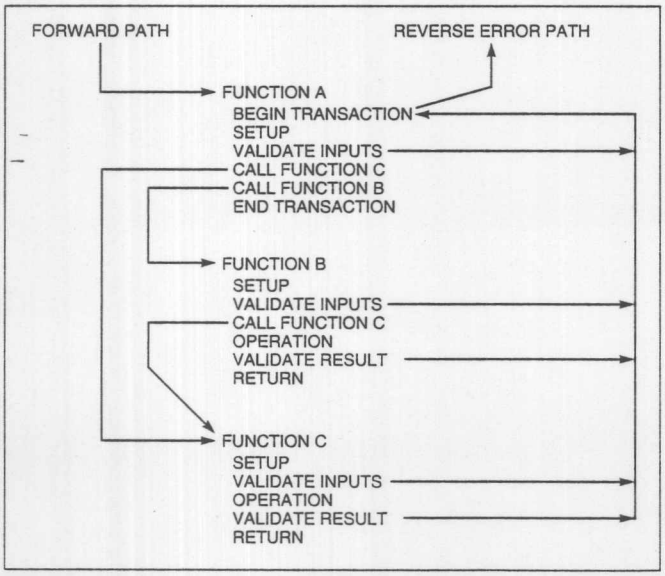

Fig 2—In transaction error-handling program flow, only error-detection code for real errors remains, and most of the error-correction and recovery code clusters around the beginning and end of the transactions.

From the perspective of *function B()* and probably *function A()*, an out-of-memory error is a tramp error.

Tramp errors prevent functions from becoming "black boxes." For example, *function A()* (above) now knows something about how *function B()* works. In other words, tramp errors form a part of error recovery—not error detection—because if the program could immediately correct real errors, tramp errors would not occur. Because of tramp errors, almost every function has to handle errors that lower-level functions generate. This buck-passing can cause tight data coupling, which makes code reuse more difficult.

Unreadable code and poor reuse

Mixed forward and reverse paths and tramp errors combine to obscure the main forward path of the program, which is doing the real work. The correction and recovery parts of error handling are the main areas that obscure the code. Most of the code for detection and reporting can be in separate functions, so these components of error handling play less of a role in obscuring code than do the other two.

You can solve the problems of unreadable code and poor reuse by separating the forward and reverse error-processing paths and by using context-independent error codes. This method of error handling is very similar to the way databases handle error recovery. Transactions control the rollback process when a group of database operations cannot complete successfully.

The traditional defensive way of programming is to assume that a function may have failed to complete its task, resulting in a lot of error-handling code to check for the errors and to roll back partially completed work, as **Fig 1** shows. Now, assume the reverse—that returning functions have successfully completed their tasks. In this scenario, if the function or one of the functions it calls has errors, the function passes processing control to a programmer-defined recovery point. In other words, the programmer defines transaction points so that if there are any problems, the work rolls back to those points, and the processing again proceeds. With this approach, you do not need to check for errors after each function call, and tramp error-detection code does not clutter the forward path.

Context-independent error codes provide more information than just an error number. They also provide information such as which function generated an error, the state that caused the error, the recommended correction, and the error severity. This information allows the program to correct the error in a location separate from the forward processing path.

Programmers usually encode contexts of errors for error-reporting functions. For example, error contexts may include the names of the program, the function, the error type, and the error code. The program saves these parameters to report later. However, programmers rarely use sophisticated encoding schemes because traditional error handling already knows the context of the error: Checking occurs right after a call to the offending function.

With transaction error handling, the recovery process is separate from the forward processing path, necessitating the use of context-independent error codes. This may involve creating unique error codes across a whole application or system (with the codes bound at compile time). An alternative

would be to assign code ranges or other unique identifiers to functions at runtime.

Code readability and reuse

The transaction error-handling approach makes programs easier to read because the reverse-processing paths are visually separate from the forward-processing paths. This method makes possible creating some general error-recovery interfaces so that functions (modules or objects) connect only loosely at the error-handling level. This loose connection is possible because the there are fewer tramp errors to control the recovery process, and the program needs to handle only the real errors.

Two methods you need for building a transaction error-handling library are transaction-control and transaction-data management. Transaction-control management requires some language support to implement the mechanism that controls error recovery. For example, languages like Hewlett-Packard Co's Pascal-MODCAL have a "try/recover" feature that can support a transaction error-handling style.

For other languages, you must use a "global-goto," or "multithreaded," feature, which allows a lower-level function and all other functions above it to exit to a point you define in a higher-level function without passing error-code flags through all the other functions. In C, you do this with the *setjmp* and *longjmp* library routines. The *setjmp* function saves its environment stack, and *longjmp* restores that environment. The **listings**, which are written in C, show how these functions work.

The material in **Ref 2** details the new C++ exception-handling feature, which provides an excellent foundation for a transaction-based error handler. The material in **Ref 3** also

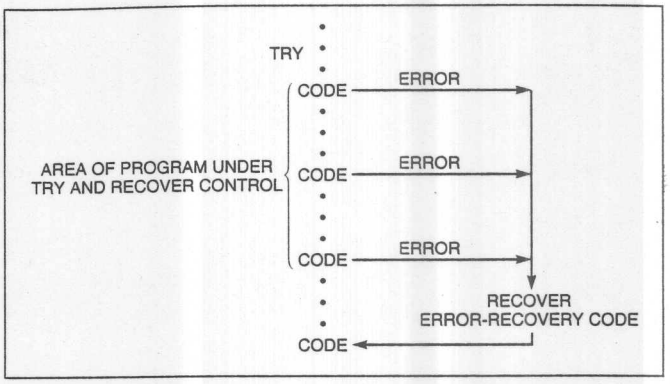

Fig 3—A try/recover statement defines error-recovery code that executes if a program detects an execution error within a particular area.

describes how to add C++ error-handling functions to regular C programs. However, overuse of transaction error handling can lead to code that is just as cluttered as the traditional error-handling style. You must design transaction boundaries for objects with the same care that you would to design an object's interface.

If a language is missing a global-goto or multithreaded feature, use macros or other "wrapper" functions to build recovery processes that are mostly invisible. Wrapper functions and macros add functionality to functions that you cannot change, such as library functions.

In building a transaction-handling package, you might want to give it the following features:

- Allow nested transactions by keeping the transactions' beginning points on a stack.
- Allow functions to share a common transaction stack.

LISTING 1 - TRADITIONAL ERROR-HANDLING STYLE

```
/*   read.c - Read a binary formatted file                          */
/*   This program reads and prints a binary file that has the       */
/* following structure:                                             */
/*                                                                  */
/*   Record type code (The last record has a value of 0)            */
/*   Size  Number of characters in Msg                              */
/*   Msg   0 to 2048 characters                                     */
/*   Record type code                                               */
/*   Size                                                           */
/*   Msg                                                            */
/*   .                                                              */
/*   .                                                              */
/*   .                                                              */

#define aExitErr(pMsg, pErr)      puts(pMsg); exit(pErr)
#define aRetErr(pMsg, pErr)       puts(pMsg); return(pErr)

typedef struct {
     long    Type;
     int     Size;
}    aFileHead;
/*                                                                  */
/*   Forward Algorithm:                                             */
/*                                                                  */
/*   Main                                                           */
/*      1. Open the file.                                           */
/*      2. Call the Read process.                                   */
/*      3. Close the file.                                          */
/*                                                                  */
main() {
     int     Err;
     FILE *  InFile;
     if ((InFile = fopen("file.bin", "rb")) == NULL) {
               aExitErr("Error: Could not open: file.bin",1);
     }
     if ((Err = aRead(InFile)) != 0) {
               aExitErr("Error: While reading: file.bin", 2);
     }
     if (fclose(InFile) {
               aExitErr("Error: Closing: file.bin", 9);
     }
}/* main() */

/*   Forward Algorithm continued:                                   */
/*                                                                  */
/*   Read Process                                                   */
/*   1. Read the Type and Size values.                              */
/*   2. If Type = 0, exit.                                          */
/*   3. Read Size number of characters into the                     */
/*      Msg variable.                                               */
/*   4. Print the Msg.                                              */
/*   5. Go to step 1.                                               */
/*                                                                  */
int aRead(pHandle)
     FILE *  pHandle;
{
     int                  Err, N;
     char *               Msg;
     long                 RecNum;
     aFileHead            RecHead;

     if ((Msg = (char *) malloc(2048)) == NULL) {
               aRetErr("Error: Out of memory", 3);
     }
     RecNum = 0L;
     while (1) {
          if (fseek(pHandle, RecNum, SEEK_SET) < 0) {
                    aRetErr("Error: in fseek", 4);
          }
          N = fread((char *) &RecHead, sizeof(aFileHead), 1, pHandle);
          if (N < 0) {
                    aRetErr("Error: in fread", 5);
          } else if (N != 1) {
                    aRetErr("Error: short fread", 6);
          }
          if (RecHead.Type == 0L) {
                   return(0);              /* EOF */
          }
          if (RecHead.Size) {
                    if ((N = fread(Msg, RecHead.Size, 1,
                       pHandle)) < 0) {
                          aRetErr("Error: in fread", 7);
                    } else if (N != 1) {
                          aRetErr("Error: short fread",8);
                    }
                    if ((Err = aPrint(Msg,
                       RecHead.Size)) != 0) {
                          aRetErr("Error: in aPrint", Err);
                    }
          }
          RecNum = RecNum + RecHead.Size + sizeof(aFileHead);
     }
} /* aRead() */
```

LISTING 2 - TRANSACTION ERROR-HANDLING METHOD

```
/*   read.c - Read a binary formatted file                          */
/*   This program reads and prints a binary file that has the       */
/*   following structure:                                           */
/*                                                                  */
/*   Record type code (The last record has a value of 0)            */
/*   SizeNumber of characters in Msg                                */
/*   Msg  0 to 2048 characters                                      */
/*   Record type code                                               */
/*   Size                                                           */
/*   Msg                                                            */
/*                                                                  */
#include"erpub.h"
#include"epub.h"
typedef struct {
     long  Type;
     int   Size;
} aFileHead;
/*                                                                  */
/*   Forward Algorithm:                                             */
/*                                                                  */
/*   Main                                                           */
/*      1. Open the file.                                           */
/*      2. Call the Read process.                                   */
/*      3. Close the file.                                          */
/*                                                                  */
main() {
     FILE *  InFile;
     erRecOn = 1;
     if (erSet()) { /* Transaction rollback point */
        printf("Error: %d in function: %s\n", erErr,
                erFun);
        erUnset();
        exit(1);
     } /* End Recovery section */

     InFile = fopen("file.bin", "rb");
     aRead(InFile);
     fclose(InFile);
     erUnset();
} /* main() */
/*   Forward Algorithm continued:                                   */
/*                                                                  */
/*   Read Process*/
/*      1. Read the Type and Size values.                           */
/*      2. If Type = 0, exit.                                       */
/*      3. Read Size number of characters into the                  */
/*         Msg variable.                                            */
/*      4. Print the Msg.                                           */
/*      5. Go to step 1.                                            */
/*                                                                  */
int aRead(pHandle)
     FILE * pHandle;
{
     char *    Msg;
     long      RecNum;
     aFileHead RecHead;
     Msg = (char *) malloc(caMsgLen);

     RecNum = 0L;
     while (1) {
          fseek(pHandle, RecNum, SEEK_SET);
          fread((char *) &RecHead, sizeof(aFileHead), 1, pHandle);
          if (RecHead.Type == 0L) {
               return;/* EOF */
          }
          if (RecHead.Size) {
               fread(Msg, RecHead.Size, 1,pHandle);
               aPrint(Msg, RecHead.Size);
          }
          RecNum = RecNum + RecHead.Size + sizeof(aFileHead);
     }
} /* aRead() */
```

- Allow functions to define their own transactions with a common transaction stack, or allow functions to define their own transaction stacks for special cases.
- Define special transaction points to handle errors in common categories. (For example, abort the whole program, restart the whole program, close all files and restart, close current file and restart, and release all memory not needed and restart.)
- Have the transaction error handling turn on and off. (When transaction error handling is off, a function returns error codes in the traditional way.)
- Define expected errors for some functions by masking out the needed errors. (You can simulate this feature by turning off transaction error handling, but then you also have to manage unexpected errors.)

Transaction-data management

Recovery involves more than just rolling back functions to undo some intermediate work. It may also involve releasing unneeded memory or changing global variables back to the values they had at the beginning of the transaction.

You can best manage memory using a mechanism similar to the mark/release memory feature in some implementations of Pascal. The mark/release procedures allow dynamic allocation and deallocation of memory in an executing Pascal program. The C functions *malloc()* and *free()*, along with a stack of pointers to track the allocated memory, provide the best features for allocating and freeing memory. With these features, you can call a mark function just before the program's transaction starting point to mark the current stack point. If a *longjmp()* goes to this recovery point, the release function is called to free any memory allocated after the mark point.

To remove pointers from the mark/release stack, you need a commit function at the end of a program transaction. The commit function indicates the successful completion of a transaction in the database context. You must also consider nested transactions, however. A simple solution would be to have each transaction keep its own mark/release stack.

LISTING 3 - MACROS AND GLOBAL DATA STRUCTURES

```
     /* erpub.h - Error Recovery Public Include file */
     #include<setjmp.h>
     /* Private Variables */
     #define vMaxEnv      5
     jmp_buf vEnv[vMaxEnv];
     int
vLevel = -1;
     /* Public Variables */
     #define cerFunNameLen 32
     #define erSet()      setjmp(vEnv[++vLevel])
     #define erUnset()    - -vLevel
     #define erRollBack(pFun, pErr, pRet)  \
             strncpy(erFun, pFun, cerFunNameLen); \
             erFun[cerFunNameLen-1] = \0;\
             erErr = pErr;\
             if (erRecOn && vLevel >= 0) { \
                  longjmp(vEnv[vLevel], pErr);\
             } else { \
                  return(pRet);\
     } int   erErr = 0;
     char    erFun[cerFunNameLen];
     int     erRecOn = 0;
```

You can roll back global and other static variables with a strategy similar to the one in the memory-management problem. Just before a transaction's beginning point, the program saves on a stack the states of all the globals that might change. This strategy allows you to nest transactions.

Context-independent error codes

The traditional error-handling style of checking error codes after each function call automatically gives errors a context. The transaction error-handling style provides this context information in another way. The biggest challenge in transaction error handling is that error codes alone are not very useful. For example, "97" could be the letter "a" in ASCII code, the digits "6" and "1" in BCD format, index 97 in a message array, the 97th error, an out-of-memory error, a disk-full error, a divide-by-zero error, or another error.

To decode an error code, a program must know the source of the error. Some information that the program may save when an error occurs includes the machine name, the program name,

ERROR HANDLING

the process number, the module name, the function name, and, of course, the error code. The program needs to send this information only when it must roll back a transaction.

The amount of information that the program saves depends on the location of the transaction-recovery point and on the runtime environment. For example, a client-server application may need more information than does a simple PC application. Each recovery point can usually find higher-level context information fairly easily. For example, the names of the machine, program, module, and function can easily pass down to a lower-level recovery point. However, a program cannot collect lower-level context information because, by the time the program gets down to that level, the function that had the error would no longer be active.

You may want to consider the following points when implementing a transaction error-handling scheme:

- Put the rollback points, if any, at the beginning of functions.
- Put error detection and default substitution at the beginning of functions.
- Put some error-detection code in the middle of functions to check intermediate values.
- Put error-detection code at the end of functions to validate the results.
- Do not put error-handling code for managing rollbacks in the middle of a function.

Traditional error-handling style

Listing 1, which reads a binary formatted file, is coded with a common error-handling style. The code would have been more cluttered without the *aExitErr()* and *aRetErr()* macros to manage the error reporting and recovery. **Listing 1** uses the simple error-recovery process: Detect error, report error, and exit. However, notice how much error-handling code is mixed in with the algorithm.

Listings 2 through **5** show an implementation of the transaction error-handling style. **Listing 2** performs the same function as the program in **Listing 1** but uses the transaction style of error handling. The functions *erSet*, *erUnset*, and *erRollBack* provide the error handling defined in the include file *erpub.h*. **Listings 3** through **5** show the support functions for the transaction error-handling method. In the main body of the algorithm, the code following the recovery sections is clearer than that in the traditional error-handling example, and there is no error-handling or recovery code mixed in with the algorithm.

However, there are some shortcomings in the support modules. For example, most of the macros should be functions, and the program should save the *vEnv* values in a linked list. Some engineers point out that the transaction implementation of *read.c* is not really shorter than the traditional implementation of *read.c* because the error-handling code simply moves from *read.c* into the support functions. But that is exactly the goal: to remove the error-handling code from most functions and encapsulate the error-handling in common shared code.

The include file *epub.h* contains wrapper macros that cause the program to call the appropriate transaction error-handling functions instead of the standard library function. For example, when invoking the standard function *fclose*, the program actually calls the function *eClose*.

Listing 3 defines macros and global data structures that form a crude error-transaction manager. The macros perform the following operations:

- **erSet.** This macro adds a rollback point to the *vEnv* (environment) stack.
- **erUnset.** This macro removes the top rollback point from the *vEnv* stack.
- **erRollBack.** This macro saves the function name and error code in a global area (*erFun* and *erErr*), and if the *erRecOn* flag is true, control passes to the rollback point defined on the top of the *vEnv* stack. If *erRecOn* is false, *erRollBack* simply returns the usual error code.

These macros are for illustration only. Thus, there are no internal checks for problems, and you should define the global data structures as static values in a library module or create a structure to collect them. This structure is passed to each of the transaction error-handling functions.

Listing 4 contains wrapper macros that cause the program to call the functions in the file *e.c* in place of the standard library functions. The functions in *e.c* behave the same as the standard library functions, but if the error transaction manager is on (*erRecOn* is true in *erpub.h*), control passes to the last defined rollback point, rather than just returning the same error code as the associated standard library function.

Using these wrapper macros makes it easier to add transaction error handling to old programs, but if you want to

LISTING 4 - WRAPPER MACROS

```
/*epub.h - Error Library Wrapper Macros (only a few are
           shown here) */
#define ceEOF          1
#define ceOutOfMem     2
#define ceReadErr      3
#define ceReadShort    4

#ifndef vInE
#define fclose(pStream)                          eClose(pStream)
#define fopen(pFileName, pType)                  eOpen(pFileName, pType)
#define fread(pPtr, pSize, pNItem, pStream) \
        eRead(pPtr, pSize, pNItem, pStream)
#define fseek(pStream, pOffset, pPrtName) \
        eSeek(pStream, pOffset, pPrtName)
#define malloc(pSize)                            eMalloc(pSize)
#endif
```

LISTING 5 - WRAPPER FUNCTIONS

```
/* e.c - Error Library Wrapper Functions (only a few are
         shown here) */
#define   vInE
#include        "epub.h"
void *  eMalloc(pSize)
        size_t pSize;
{
        void *  Mem;
        if ((Mem = malloc(pSize)) == NULL) {
                erRollBack("malloc", ceOutOfMem, Mem);
        }
        return(Mem);
} /*    eMalloc */
size_t  eRead(pPtr, pSize, pNItem, pStream)
        char * pPtr;
        size_t pSize, pNItem;
        FILE * pStream;
{
        size_t Num;
        Num = fread(pPtr, pSize, pNItem, pStream);
        if (feof(pStream)) {
                erRollBack("fread", ceEOF, Num);
        } else if (Num <= 0) {
                erRollBack("fread", ceReadErr, Num);
        } else if (Num < pNItem) {
                erRollBack("fread", ceReadShort, Num);
        }
        return(Num);
} /* eRead   */
```

ERROR HANDLING

make the error-handling process more visible, have the program call the functions in *e.c* directly instead of the standard library functions. The file in **Listing 4** is also a good place to define context-independent error codes.

The file in **Listing 5** contains the implementations of the wrapper macros in *epub.h*. **Listing 5** shows only two of the functions. These functions behave exactly like the standard library functions with the same name because they call the standard library functions. For more flexibility, a real error transaction manager might allow you to define the error codes that determine whether a rollback occurs.

So far, only small programs and enhancements of existing programs have used the transaction error-handling technique. But, its features—greater code reuse, greater code supportability, and better quality code—may make it more widespread. Just as you can separate the functional part of algorithms from user interfaces (client/server models), you can also separate error handling from the functional algorithm. EDN

References

1. Page-Jones, M, *The Practical Guide to Structured Systems Design*, Yourdon Press, 1980, pg 104.
2. Stroustrup, B and M Ellis, *The Annotated C++ Reference Manual*, Addison-Wesley Publishing Co, 1990.
3. Vidal, C, "Exception Handling," *The C Users Journal*, September 1992.

Acknowledgments

Thanks to Andra Marynowski and Kevin Wentzel, coworkers at Hewlett-Packard, who helped review and refine the ideas in this article, and to King Wah Moberg for providing a number of reviews.

Author's biography

Bruce A Rafnel is a software engineer at the Professional Services Division of Hewlett-Packard Co, Mountain View, CA, where he has worked for 12 years. In his current position, he develops Standard General Markup Language (SGML) applications that help deliver HP's customer-training courses. Rafnel also helped develop the Charting Gallery graphics package for PCs and internal SGML applications, and he helped enhance the VPlus forms manager for HP3000 Unix systems. He earned a BS in computer science at California Polytechnic State University at San Luis Obispo. A member of the IEEE and the C Users Group, Rafnel lists voice-controlled home automation as one of his spare-time interests.

Article Interest Quotient (Circle One)
High 590 Medium 591 Low 592